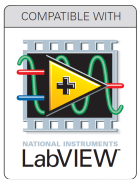
Smart phone & Cross-platform Communication Toolkit User Manual



Release 1.0.1
July 2011 Edition

Revision 1.0.1.70

Worldwide technical support and product information:
www.toolsforsmartminds.com
TOOLS for SMART MINDS Corporate headquarter
Via Padania, 16 Castel Mella 25030 Brescia (Italy)

Copyright © 2010 Tools for Smart Minds. All rights reserved.

Contents

FIGURE INDEX 6

ABOUT THIS MANUAL 7

CONVENTIONS 7

INTRODUCTION 8

OVERVIEW 8
TOP REASONS TO USE SMARTPHONE & CROSS PLATFORM COMMUNICATION TOOLKIT (SCCT) 8
SCCT APPLICATION EXAMPLES 9
COMMUNICATING DATA ACROSS A NETWORK, ON A COMPLEX ASSEMBLY LINE 9
PUBLISHING REAL TIME DATA TO MOBILE DEVICES 9
DELIVERING HIGH QUALITY MAINTENANCE SERVICE TO YOUR CUSTOMER 10
INSTALLING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 10
SUPPORTED PLATFORMS 11
PUBLISHER LIBRARY 11
SUBSCRIBER LIBRARY 11

GETTING STARTED WITH THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 12

COMMUNICATION CONCEPTS 12
USING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 14
PUBLISHER CLASS 14
CREATING A PUBLISHER 15
PUBLISHING ANALOG DATA 15
DEFINING A SYSTEM CONFIGURATION 16
PUBLISHING DIGITAL DATA 17
SUBSCRIBER CLASS 18
CREATING A SUBSCRIBER 19
STARTING AND STOPPING DATA TRANSMISSION 19
SENDING REQUESTS 21
RECEIVING MESSAGES 21
HOW TO CHECK THE STATE OF YOUR CONNECTION 22
CLOSING COMMUNICATION 22

CREATING AND MANAGING ALERTS AND USER REQUESTS 23

OVERVIEW 23
READING AVAILABLE REQUESTS 23
SENDING MESSAGE TO A SPECIFIC SUBSCRIBER 23
MANAGING USER REQUESTS 24
NOTIFYING AN ERROR 24

ADVANCED OPTIONS 25

CONTROLLING ACTIVE CONNECTIONS 25
READING ACTIVE CONNECTION COUNT 25
READING ACTIVE CONNECTION ADDRESSES 25
PUBLISHING DIGITAL LINES ON CHANGE ONLY 26
MANAGING EVENTS 27
CHANGING API-KEY AT RUN-TIME 28
CHANGING MAX DT 29
PUBLISHING AND RECEIVING CUSTOM DATA (SUPPORTED ONLY BY LABVIEW APPLICATIONS) 30
PUBLISHING AND RECEIVING CUSTOM XML DATA 31

APPENDIX A - ERROR TABLE 33

PUBLISHER ERROR CODES 33
SUBSCRIBER ERROR CODES 33
INDEX **34**

Figure index

FIGURE 1 – HOW DATA PACKAGES ARE ORGANIZED AT SUBSCRIBER SIDE. 12
FIGURE 2 – AN APPLICATION CAN PROCESS INCOMING PACKAGES ACCORDING TO THEIR DATA TYPES. 13
FIGURE 3 - SIMPLE PUBLISHER EXAMPLE WITH ANALOG DATA. 15
FIGURE 4 - SIMPLE PUBLISHER EXAMPLE WITH DIGITAL DATA. 17
FIGURE 5 - SIMPLE SUBSCRIBER EXAMPLE THAT READS ANALOG DATA ONLY. 19
FIGURE 6 – TRANSFERSTATUS.VI DETERMINES THE FLOW OF PACKETS BETWEEN PUBLISHER AND SUBSCRIBER. 19
FIGURE 7 - SIMPLE SUBSCRIBER EXAMPLE THAT READS DIGITAL DATA ONLY. 20
FIGURE 8 - SIMPLE SUBSCRIBER THAT READS CONFIGURATION CLUSTER BEFORE READING ANALOG DATA. 20
FIGURE 9 - SENDREQUEST.VI EXAMPLE. 21
FIGURE 10 - READING A MESSAGE FROM PUBLISHER. 22
FIGURE 11 - USING PROPERTIES NODES TO CHECK CONNECTION STATUS. 22
FIGURE 12 - USING AVAILABLE REQUEST PROPERTY NODE TO CHECK IS SUBSCRIBER'S REQUESTS ARE PRESENT. 23
FIGURE 13 - SENDING MESSAGE TO A SPECIFIC SUBSCRIBER WITH SENDMESSAGE.VI. 23
FIGURE 14 - MANAGING RECEIVED REQUEST EXAMPLE. 24
FIGURE 15 - NOTIFYING LABVIEW ERROR TO ALL SUBSCRIBERS WITH NOTIFYERROR.VI. 24
FIGURE 16 - USE READ ACTIVESUBSCRIBERSCOUNT.VI TO CHECK THE COUNT OF ACTIVE CONNECTIONS. 25
FIGURE 17 - USE READ ACTIVESUBSCRIBERSADDRESSES TO GET INFO ABOUT ACTIVE SUBSCRIBERS. 25
FIGURE 18 - USE PROPERTY NODES TO GET INFORMATION ABOUT ACTIVE SUBSCRIBERS. 25
FIGURE 19 – CHANGING THE WAY DIGITAL DATA ARE UPDATED TO ALL SUBSCRIBERS WITH DIGITAL.PUBLISHONLYONCHANGE PROPERTY NODE. 26
FIGURE 20 – ENABLING THE REQUEST FIFO WITH ENABLEEVENTLOGGING PROPERTY NODE. INSTEAD OF PROPERTY NODE YOU CAN SET THE FLAG "ENABLELOGGING" WITH **ENABLELOGGING.VI** AND GET ITS VALUE WITH **ISEVENTLOGGINGENABLED.VI**, RESPECTIVELY. 26
FIGURE 21 – ENABLING THE REQUEST FIFO WITH ENABLEEVENTLOGGING PROPERTY NODE. INSTEAD OF PROPERTY NODE YOU CAN SET THE FLAG "ENABLELOGGING" WITH **ENABLELOGGING.VI** AND GET ITS VALUE WITH **ISEVENTLOGGINGENABLED.VI**, RESPECTIVELY. 27
FIGURE 22 – THIS EXAMPLE SHOWS HOW TO MANAGE "AUTHENTICATION SUCCESSFUL" EVENT FOR INSTANCE, NOTICE THAT WELCOME MESSAGE IS ADDRESSED TO NEW CONNECTED SUBSCRIBER AND OTHER ACTIVE SUBSCRIBERS ARE NOT AFFECTED BY THIS MESSAGE.. 28
FIGURE 23 – THIS EXAMPLE SHOWS HOW TO MANAGE THE CASE A CONNECTION IS NO MORE ACTIVE. IT CAN HAPPEN WHEN CLIENT CLOSES THE CONNECTION OR SOME ERRORS OCCUR DURING DATA TRANSFER OR TIMEOUT, EXPIRES. 28
FIGURE 24 – CHANGING API-KEY TO AN ACTIVE PUBLISHER WITH WRITE API-KEY.VI. 28
FIGURE 25 – CHANGING API-KEY TO AN ACTIVE PUBLISHER USING API-KEY PROPERTY NODE. 29
FIGURE 26 – CHANGING MAXDT TO CHECK SUBSCRIBER'S SYSTEM TIME. 29
FIGURE 27 – PUBLISHER APPLICATION AND SUBSCRIBER APPLICATION USE CODE FIELD TO IDENTIFY THE PROPER DATA TYPE. BOTH APPS HAVE TO KNOW THE DATA TYPE ASSOCIATED TO THE CODES. 30
FIGURE 28 – WITH PUBLISHER, USE PUBLISHDATA.VI METHOD TO SEND XML DOCUMENTS. 31
FIGURE 29 – WITH SUBSCRIBER, USE SENDCUSTOMXMLDATA.VI TO SEND XML DOCUMENTS TO PUBLISHER. 31
FIGURE 30 – WITH PUBLISHER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS. 31
FIGURE 31 – WITH SUBSCRIBER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS. 31
FIGURE 32 – WITH PUBLISHER AND SUBSCRIBER YOU HAVE DIFFERENT METHODS AND PROPERTY NODES TO GET THE COUNT OF AVAILABLE XML UNPROCESSED DOCUMENTS. 32

About this Manual

The Smartphone & Cross-platform Communication Toolkit User Manual describes the virtual instruments (VIs) used to communicate and pass data between LabVIEW and either a local or remote application. You should be familiar with the operation of LabVIEW, your computer and your computer operating system.

Conventions

The following conventions appear in this manual:

▸	The ▸ symbol leads you through nested menu items and dialog box options to a final action. The sequence **Tools** ▸ **Options** directs you to pull down the **Tools** menu, select **Options** item.
Bold	Bold text denotes items that you must select or click on the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Introduction

This chapter describes the installation procedure, installed components, and the main features of the Smartphone & Cross-platform Communication Toolkit.

Overview

The Smartphone & Cross-platform Communication Toolkit is an add-on package for communicating data trough applications. The toolkit contains a set of high level functions for sending your application data and advanced functions for customized tasks.

The following list describes the main features of the Smartphone & Cross-platform Communication Toolkit:

- Works over any TCP/IP connection
- Works over Local Area Networks as well as Internet connections.
- Implements the publisher – subscriber pattern (also known as Observer pattern)
- Authenticates subscribers through a API-KEY.
- Controls in background the state of every connection to identify loss of communication.
- Publishes GPS coordinates to manage mobile systems.
- Works with platform independent Data format and communicate with multiple platforms at the same time: third party vendors have implemented toolkit to develop on Android platform, Java, .NET and VB, Unix and iOs.

Top reasons to use Smartphone & cross platform Communication Toolkit (SCCT)

Adopting this toolkit you have the following advantages:

Simplify communication: don't care about communication details over a TCP communication channel, SCCT does it for you.

Multiple platforms are supported: exchange your data with a protocol supported on a wide range of platforms and programming languages.

It's reliable: many applications have been created with this toolkit around the world.

Speed up your development activity: this toolkit allows the creation of distributed application and let you save a lot of your time.

SCCT has been created by LabVIEW developers for LabVIEW developers: it includes some great features supported on LabVIEW platform only (see Publishing CustomData in this manual) so if you need to exchange data with other LabVIEW applications, take

advantage of the power of SCCT to deliver high quality code and reduce developing time.

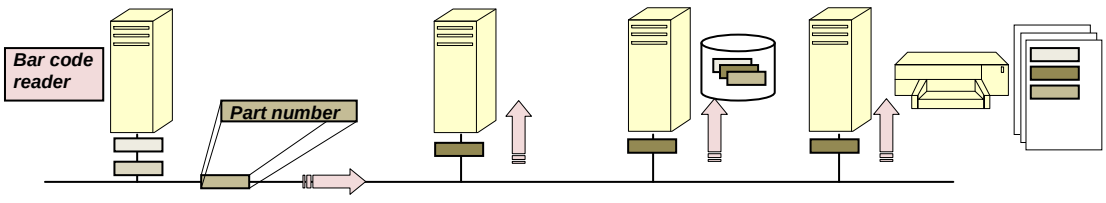
Today applications need to retrieve information from database, bar code readers, OCR systems, remote data acquisition or technical operators with mobile devices. Often you have to create systems capable to exchange data with legacy applications created on different platforms. Every time you have to design an ad hoc communication protocol, code both server and client side routines, debug them. Every time! With SCCT you have a new tool that let you save a lot of time, lets you focus on your project's core and manage multiple communications at the time. In the following examples you are going to see how SCCT can dramatically improve the quality of your software solutions.

SCCT application examples

SCCT can be successfully applied in many real world situations. In this chapter we discover where SCCT helps developers to succeed to deliver high valued solutions.

Communicating data across a network, on a complex assembly line

Creating an application which broadcasts part numbers and barcodes coming from databases or bar code readers, to all computers on a assembly line, as schematized below. On modern assembly lines, many computers control single processing stations and they need to exchange data among them to know, part numbers, print codes and certificates, store test results on different databases. SCCT provides many functionality to exchange data with applications developed with heterogeneous programming languages. besides, when your customer asks you to show some data on a mobile device, SCCT is capable to communicate with a large variety of smart phones and tables and you don't have to change a single line of code of your software. SCCT cares of all connected devices and communicate your data to all of them at the same time, receives user's requests and organize them in a time-ordered FIFO so that you can process them easily.



Publishing real time data to mobile devices

When you have to publish acquired data in real time, SCCT is the best choice because with few SCCT Vis your applications is supported on an wide range of platforms: Android, iPhone, etc. SCCT is safe because every device must authenticate to your application and you can control who is connected in any time. Using SCCT you deliver open solution to your customers because they can use their favorite mobile platform to connect to your data. And if customers change mobile device family, you don't need to modify your code!

Delivering high quality maintenance service to your customer

Including SCCT into your existing applications, let you offer fast support to your customers. You and your customers can monitor deployed applications everywhere, with a tablet, phone, desktop. integrating SCCT capabilities into existing applications makes easy debug and signal analysis of deployed systems around the world.

Installing the Smartphone & Cross-platform Communication Toolkit

Smartphone & Cross-platform Communication Toolkit is shipped as a VI Package Manager. You can download it from

www.toolsforsmartminds.com/products/SCCT.php

Before installing Smartphone & Cross-platform Communication Toolkit you must install a copy of VI Package Manager on your machine. You can get a free copy of VIPM at this address:

http://www.jki.net/vipm/download

To install Smartphone & Cross-platform Communication Toolkit double click open the vip file:

smartphone_&_crossplatform_communication_toolkit-x.x.x.xx.vip

and follow the installation wizard. Package contains LabVIEW Vis as well as documents in PDF format accessible from LabVIEW (**Help ▸ TOOLS for SMART MINDS ▸ SCCT User Guide)** and libraries to create applications with Java, Android and .Net (in *c:\SCCT\cross-platform libs*). Visit http://www.toolsforsmartminds.com to get more details.

Supported platforms

SCCT is composed by two main components:

- publisher library
- subscriber library

publisher library

this library let you create a full-featured publisher, which authenticates incoming subscribers, check connection status, sends data to all active publishers and passes their request to your application.

This library is available as a set of Vis for LabVIEW 2010 or later.
To get more details or download an evaluation copy of this library please visit:

http://www.toolsforsmartminds.com/products/SCCT.php

Subscriber library

This library let you create a subscriber which handles all communication details with a publisher so you don't have to. It receives data packages and present them to your application according to their data types.

This library is available for the following platforms and languages:

Name	***Operating System***	***Development Language***
SCCT Subcriber for LabVIEW[1]	Windows	LabVIEW 2010
SCCT Subscriber for VB[2]	Windows	Visual Basic, .NET,
SCCT Subscriber for Java	Java VM 5.0 or later	Java
SCCT Subscriber for Android	Android 2.1 or later	Java
SCCT Subscriber for Linux[3]	Linux Kernel 2.6.30	ANSI - C
SCCT Subscriber for iPhone/iPad[4]	iOs	Objective C
SCCT Subscriber for Phone7[5]	Phone7	.NET

To get more details or download your **free** copy of SCCT subscriber library, please visit:

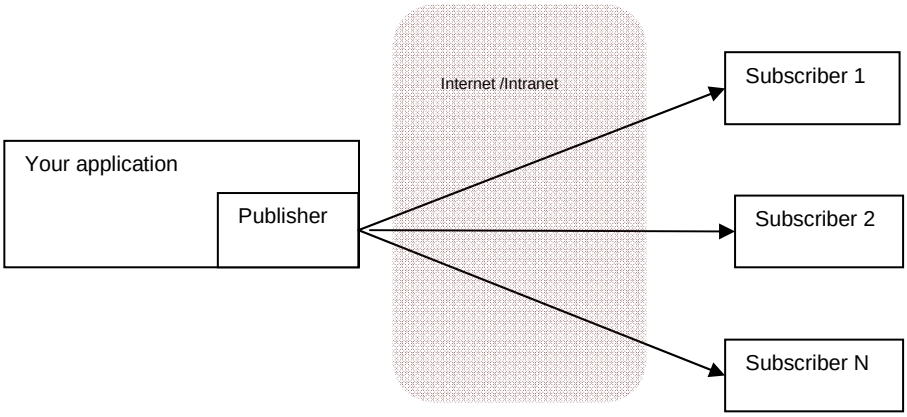
http://www.toolsforsmartminds.com/products/SCCT.php

[1] This library is included in SCCT VIPM package and is not available separately from SCCT Publisher library.
[2] This library is distributed as OCX component and can be used by any language that support ActiveX technology.
[3] This library will be available from august 2011. this library works on x8
6 32bit and ARM ver. 11 processor families
[4] This library will be available from Q4 2011.
[5] This library will be available from Q4 2011.

Getting started with the Smartphone & Cross-platform Communication Toolkit

Communication concepts

Smartphone & Cross-platform Communication Toolkit is based on the publisher-subscriber pattern (also called *Observer* pattern). In this pattern you have one application (publisher) that receives the data you want to publish, and one to many applications (subscribers) which subscribe the service. To subscribe the service and receive fresh data from the publisher, they must use an API-KEY to be authenticated. The following figure represents the pattern:



When an application wants to receive data, first asks the publisher to be inscribed among the active subscribers. Publisher will accept all incoming requests with the valid API-KEY.

Received data are organised into separated FIFOs so that subscriber application can process data packages according to their types. In the following figure it's shown the case some different type packages are published according to server logic and subscriber side task organizes the nine received packages into different FIFO structures.

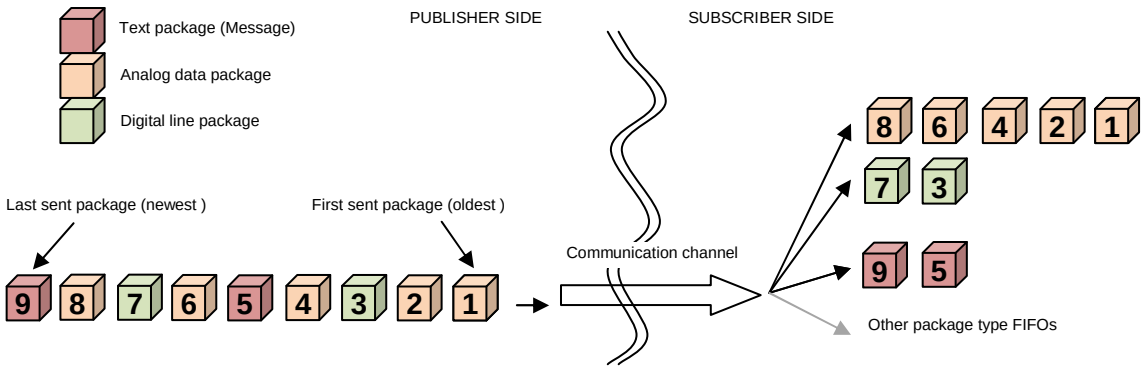


Figure 1 – How data packages are organized at subscriber side.

In this architecture, packages can be processed very easily with dedicated Vis. In many real life applications the exact order in which data packages are generated is not very important, and with LabVIEW multithread capability you can dedicate specific task to every package type and deliver robust software solution as shown below.

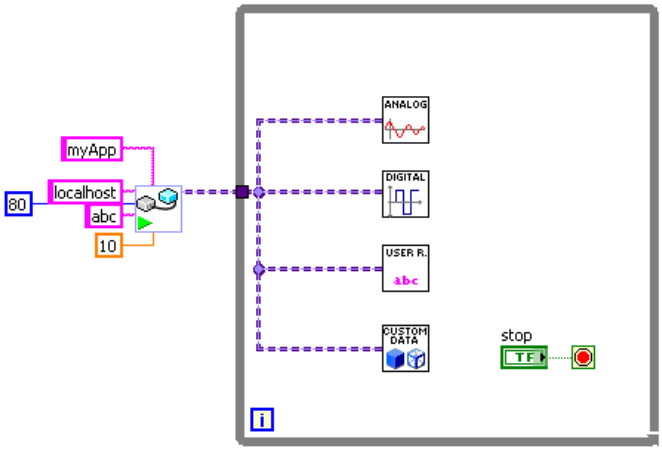


Figure 2 – An application can process incoming packages according to their data types.

Because of the wide range of devices the Smartphone & Cross-platform Communication Toolkit works with, some portability issues remain. Consider the following issues when choosing your way to publish data:

- Some smart phones and tables uses CPU with low computing power so are not able to receive and process large streams of data.
- Smartphone & Cross-platform Communication Toolkit uses a platform independent data format and subscribers require a some computing power to decode data streams into their specific binary format.
- Smartphone & Cross-platform Communication Toolkit handles communication with subscribers as a set of peer to peer connections and every data you publish is transmitted individually to each subscriber. So you have to identify the right size of your data streams to avoid band saturation over your communication channel.
- Some data types are not supported on all platforms.

Using the Smartphone & Cross-platform Communication Toolkit

This Toolkit is composed of two main components: Publisher that creates the server side of your communication system and the subscriber that implements the client side. Publisher and Subscriber work together to pass data from one application which holds the data to many applications on different systems (MS-Windows OS family, Linux, Apple systems, mobile devices, etc.). Publisher uses a platform independent data format to transmit your data so that all subscribers can read them. Doing so you add a little overhead to a simple transmission that uses binary data format, but gain a great portability and opportunity to communicate with heterogeneous systems. To better understand the way this communication works, think of this example. A publishing company receives requests from different subscribers who want to receive a magazine. As long as they are subscribed, they receive the magazine. When they don't want to receive it anymore, simply cancel their subscription. Your application can implement more than one Publisher each of them works on different port of your machine. An application can contain publishers and subscribers together, working with different remote machines at the same time.

Either objects work in background of your application with specific tasks that are created and destroyed automatically. This library has been created to LabVIEW developers and includes some advanced features supported only by LabVIEW apps. Please be careful when design your communication solutions so that your data can be properly treated by all subscriber. So when your system is designed to work with LabVIEW programs only, customData packages greatly simply data transfer among processes, but cannot be processed by subscribers created, for example, with SCCT for Java.

Publisher Class

Publisher is a Class with methods and properties detailed in the following tables:

Publisher Class properties		
Property name	*description*	*Read/write*
availableRequests	gets the count of received requests from subscribers	Read only
enableEventLogging	Sets/gets the management of subscriber requests.	Read/Write
Port	Gets the actual port number	Read only
API-Key	Gets the actual API-Key	Read/Write
activeSubscribersCount	Gets the count of active connections	Read only
activeSubscribersAddresses	Gets the IP address and port of each active connection	Read only
Digital.publishOnlyOnChange	Defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.	Read/Write
maxDT	Sets/gets the maximum allowed time difference between publisher time clock and subscribers time click.	Read/Write

Publisher Class methods	
method name	*description*
PublishData	Publishes analog and/or digital lines
startPublisher	Implements a publisher object
stopPublisher	Destroy a Publisher object
updateConfiguration	Transmit a new system configuration to all active connections
getRequest	Get the first unprocessed request from one of the active connections
getEvent	Get the first unprocessed event.
sendMessage	Transmit a message to a specific connection or to all active connections
notifyError	Transmit a message to a specific connection or to all active connections with a

	LabVIEW error code and source.
Read availableRequestCount	Returns the count of unprocessed requests.
Read activeSubscriberCount	Returns the count of active subscribers
Read activeSubscriberAddresses	Returns the IP address and TCP port of active subscribers
Read Port	Returns the current TCP port used by Publisher
enableEventLogging	Enables the management of requests..
Read API-Key	Returns current API-Key
Write API-Key	Changes the current API-Key. Active connections are not affected by this change.
Read publishDigitalDataOnlyOnChange	Returns the property that defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.
Write publishDigitalDataOnlyOnChange	Sets the property that defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.

Creating a Publisher

To create a publisher in your LabVIEW application use the palette **SCCT ▸ Publisher**. The following examples show how to create a publisher in few click.
To create a publisher in your application you must choose two parameters that have to shared with subscribers:

- Publisher port is the TCP port that Publisher uses to manage all TCP connections.
- API-Key is the connection password that subscribers must communicate to publisher to be authenticated.

Take care to use one of the available port of your machine. Some ports are reserved for other common applications like port 21 to FTP, 80 to HTTP and so on. Moreover you have to check that the chosen port is open on your company firewall.

Publishing analog data

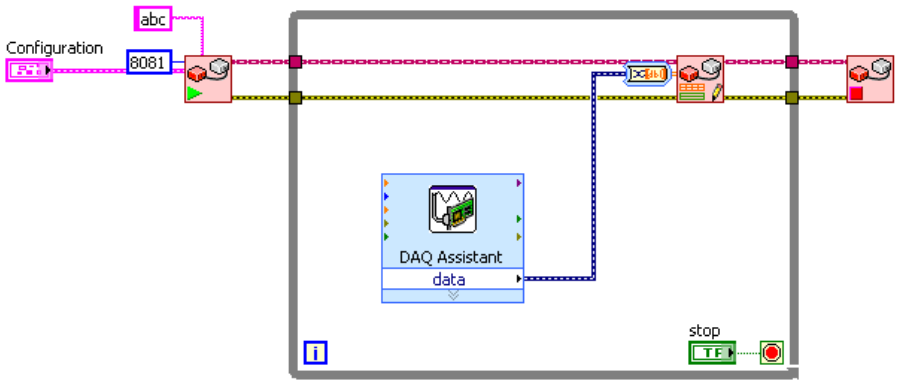


Figure 3 - Simple Publisher example with analog data.

In the example above, a Publisher is created with **startPublisher.vi** that immediately creates all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the analog signals you want to transmit. channelConfiguration must describe each channel of your data acquisition.
In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded.
When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

Notice: ***startPublisher.vi*** *create some background tasks which handle the data transfer to and from active subscribers. These tasks are also responsible of checking if connections are lost in case your application is not publishing any information. All these tasks periodically check if your top level VI is running and stop automatically when your top level VI stops.*

Defining a system configuration

Publisher requires a system configuration cluster with at least one analog channel description or one digital line description. If you connect an empty configuration cluster, startPublisher.vi rises up an error and doesn't start. The reason of this is behavior because, in past, SCCT was used to publish data from acquisition boards. Now, many new features are available and SCCT can be used to transmit a wide range of different data types, not related to any data acquisition, but configuration constrain remains to keep compatibility with legacy code.

Publishing digital data

In this example, a Publisher is created with **startPublisher.vi** that immediately create all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the digital lines you want to transmit. digitalLines.Configuration must describes each line of your data acquisition. In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded. When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

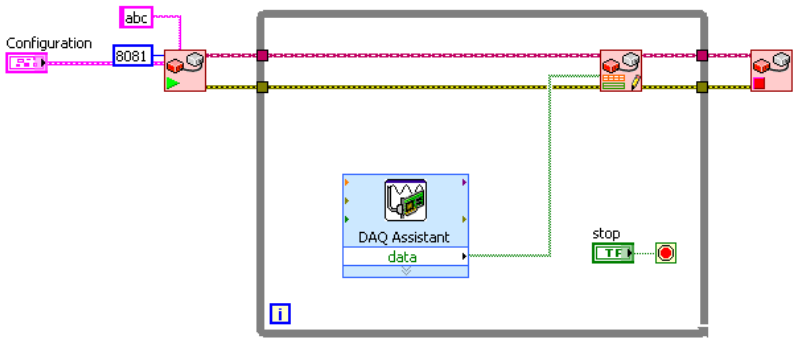


Figure 4 - Simple publisher example with digital data.

Please note in the above figure, that when you start a publisher you create a publisher object with a small set of parameters. The new object will manage by itself most of communication details. If you want to modify the way communications are handled, use LabVIEW property nodes to set desired values to publisher options. The time that occurs between the execution of publishData.vi and the effective transmission of data package to all active subscribers is usually few milliseconds and depend mostly on data size not on the active connection number.

Subscriber Class

Subscriber is a Class with methods and properties detailed in the following tables:

Subscriber Class properties		
availableMessageCount	*Gets the number of received alerts from Publisher that application has to process*	*Read only*
availableAnalogData	Gets the number of analog data packets received from Publisher that application has to process	Read only
availableConfiguration	Gets the number of configuration clusters received from publisher that application has to process. Remember that when you establish a connection, you receive immediately a configuration cluster.	Read only
availableDigitalData	Gets the number of digital data packets received from Publisher that application has to process	Read only
connected	Returns TRUE if connection is active, FALSE if connection is lost	Read only
connectionStatus	Gets a numeric code related to connection status	Read only

Subscriber Class methods	
method name	*description*
openConnection	Creates a subscriber and open a connection with a running Publisher
closeConnection	Destroys a subscriber and close connection, if active
Read Message	Reads next available alert received from Publisher
Read analogData	Reads next available analog data packet received from Publisher. To activate data transmission, you must use **transferStatus.vi**
Read digitalData	Reads next available digital data packet received from Publisher. To activate data transmission, you must use **transferStatus.vi**
Read configuration	Returns the first unprocessed configuration cluster received from Publisher. After a connection is established, Subscriber receives a remote system configuration with a description of analog channels and digital lines, location and other system information.
Read customXMLData	Returns the first unprocessed custom XML data received from Publisher.
Read customData	Returns the first unprocessed custom data cluster received from Publisher. Your application must be able to match the custom data received (a variant value) with the code specified by the application that sends the item.
discardData	Throws all received alerts, analog and digital data and configuration clusters away
transferStatus	Set the status of transmission. If you connect TRUE, publisher start sending analog data and digital lines as soon as they are available on server side. If you connect FALSE, Publisher stop data transmission. Alerts from publisher cannot be stopped.
Send Request	Sends immediately a request to the Publisher.
Send CustomXMLData	Sends immediately a custom XML data to Publisher.
Read availableAnalogDataCount	Returns the count of unprocessed analog data packets.
Read availableDigitalDataCount	Returns the count of unprocessed digital data packets.
Read availableConfigurationCount	Returns the count of unprocessed configuration cluster packets
Read availableMessageCount	Returns the count of unprocessed messages
Read availableCustomDataCount	Returns the count of unprocessed custom data cluster packets
Read availableCustomXMLDataCount	Returns the count of unprocessed custom XML data cluster packets
Read connected	Return a Boolean value with the connection status. If TRUE then connection is active otherwise is FALSE.
Read connectionStatus	Returns a string that indicates the connections status or the failure reason in case openConnection doesn't succeed to establish a valid connection.

To implement the subscriber in your application and receive data from a source, you must know three parameters:

- Data source address, that is usually the IP address of the machine where Publisher is running on.
- Data source port, that is the TCP port of the Publisher.
- API-Key is the key necessary to be authenticated by publisher. If a subscriber uses a wrong API-Key connection is refused by publisher.

When your application succeeds to connect, publisher sends immediately a configuration of remote system i.e. analog channel descriptions, unit of measure and range of all signals, digital line descriptions and system location GPS coordinates.

Creating a Subscriber

In the following example you create a simple subcriber with **openConnection.vi** that need three parameters: Publisher address (default value is localhost), Publisher port (default value in 8081) and API-Key.

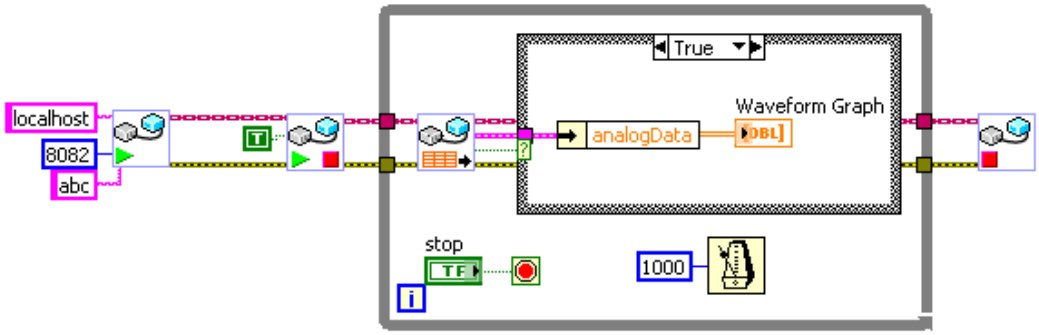


Figure 5 - Simple Subscriber Example that reads analog data only.

Publisher doesn't buffer data if transmission is stopped. The following diagram illustrates how analog and digital data are managed in situation.

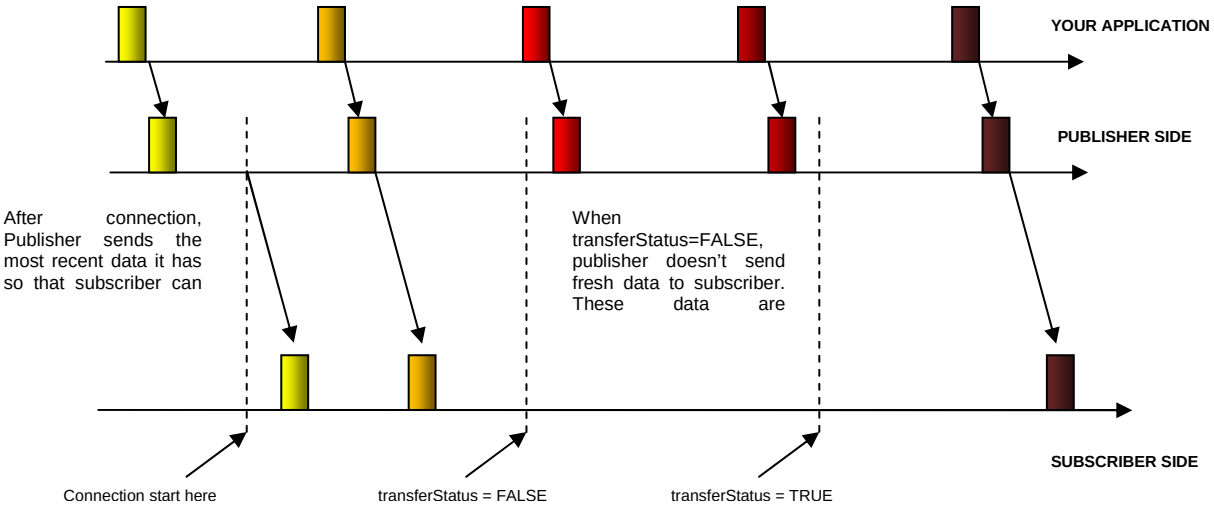

YOUR APPLICATION
PUBLISHER SIDE
After connection, Publisher sends the most recent data it has so that subscriber can
When transferStatus=FALSE, publisher doesn't send fresh data to subscriber. These data are
SUBSCRIBER SIDE
Connection start here
transferStatus = FALSE
transferStatus = TRUE

Figure 6 – transferStatus.vi determines the flow of packets between publisher and subscriber.

Starting and stopping data transmission

Publisher doesn't start data transmission automatically. After connection, your subscriber must tell to the Publisher to start sending data. If your subscriber doesn't need fresh data, use **transferStatus.vi** with a FALSE constant to tell to publisher to

stop sending data. To re-start data transmission use **transferStatus.vi** with TRUE constant.
To get analog data use **Read analogData.vi** that returns a packet of data. Background tasks takes care of all received packets and enqueues them in a FIFO so you can process all packets without data loss. To know the number of available packet use the property node that returns the data packets count. The following figure illustrate the case of reading digital lines only. Notice that at the end of the while loop you always have to close communication with publisher. When communication is closed, if you want to open again the communication use **openCommunication.vi.**

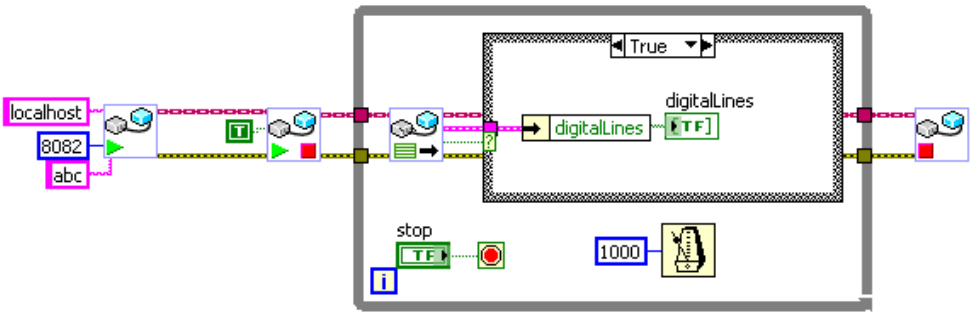


Figure 7 - Simple subscriber example that reads digital data only.

Publisher automatically sends a communication cluster that describes the remote system. Use this cluster to properly format your graph and chart setting x and y scales, as shown in the following figure.

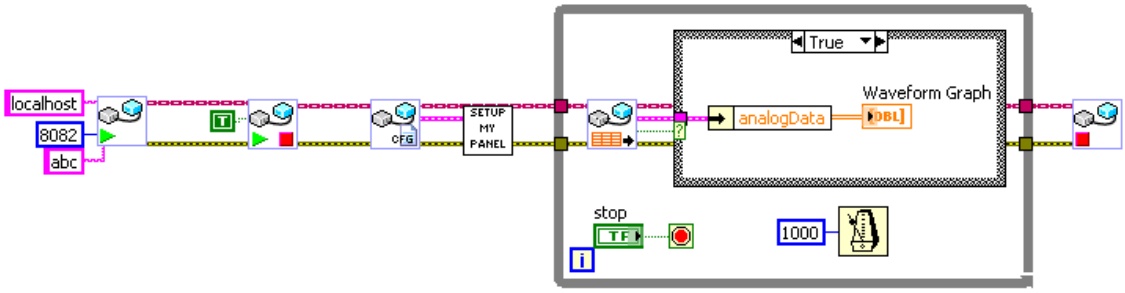


Figure 8 - Simple Subscriber that reads configuration cluster before reading analog data.

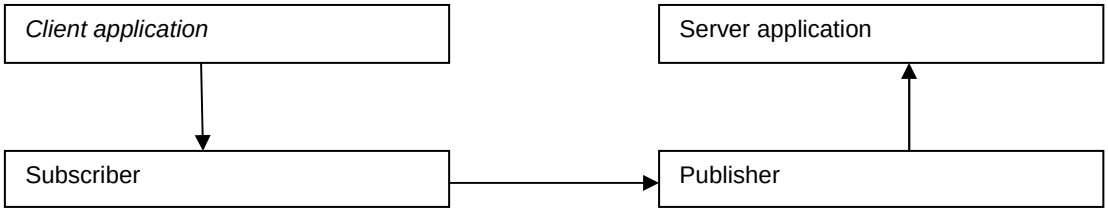
*Notice: **openCommunication.vi** create a background task which handles the data transfer to and from the publisher. This task is also responsible of checking if connection is lost in case publisher is not sending any information. This task periodically checks if your top level VI is running and stops automatically when your top level VI stops.*

Optionally you can add an application's description to the openConnection.vi. Server side application will use the description to identify your application. This is useful when the server has to properly identify subscriber's name, but they change IP address or port at every new connection, typically when DHCP is used to assign IP addresses. If description is not specified, server will identify the connection with the string

“xxx.xxx.xxx.xxx:yyyy”, xxx.xxx.xxx.xxx is subscriber application IP address and yyyy is local port used to connect to server.

Sending Requests

Subscribers can send textual requests to the Publisher with **sendRequest.vi.** In this context user means your application and a request is a string message that Publisher receives and passes directly to its main application. The following figure illustrates the path of user requests.



Requests are sent immediately to publisher and server processes them in the same order they are received. If string is empty, message is not posted. You can specify an additional numeric code if your application uses numeric codes to identify request sets.

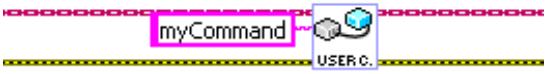
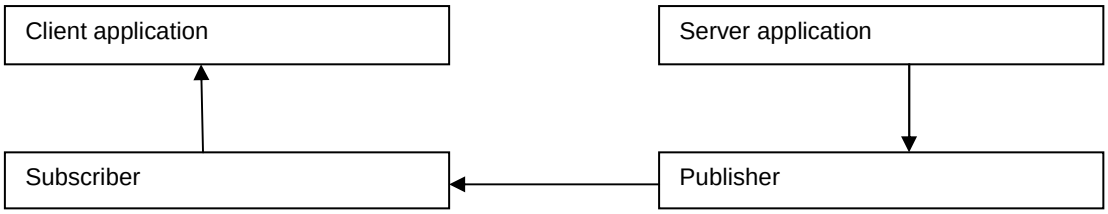


Figure 9 - sendRequest.vi example.

Receiving Messages

Subscribers can receive textual messages from the Publisher with **Read message.vi.** a message is a cluster composed by a timestamp, a string and a numeric code. The following figure illustrates the path of alerts.



Messages are received and enqueued in a dedicated FIFO. Your application have to process incoming messages and take care of FIFO size. You can use Read Message.vi to extract from the message FIFO the oldest received message.

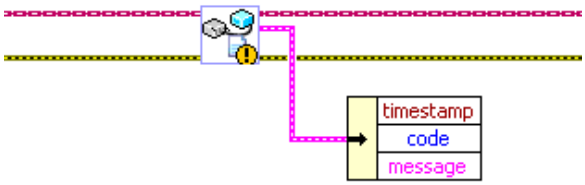


Figure 10 - Reading a message from Publisher.

activeSubscriberAddresses property node returns a 2D string array: every row contains IP address, port and description of an active subscriber. Rows are ordered by connection start time so first row regards the oldest active connection and last row regards the most recent connection.

How to check the state of your connection

You can monitor connection state with **connected** property node which returns TRUE if your connection is still alive. A connection is alive also if Publisher is not sending data to your subscriber. Publisher and subscriber exchange acknowledge packets to verify if connection is still active so you don't have to.

Figure 11 - Using properties nodes to check connection status.

Closing communication

When connection is no more necessary use closeCommunication.vi to close the open connection. This VI destroys all unprocessed data and closes background tasks. After this VI, subscriber object cannot be used and a new instance must be created with openCommunication.vi.

Creating and managing alerts and user requests

Overview

When an active communication is established between Publisher and subscriber, they can exchange some messages with a specific format: messages from your application to subscriber(s) are called Alerts. Every alert is composed by a numeric code and a message string. Messages from subscribers to your application are called user requests. Every user request contains a numeric code, a timestamp, a connection identifier, an event code and an optional data string. Publisher uses a FIFO to enqueue all incoming user requests in the order they are received. Your application can identify which subscriber is sending the request by its ipAddress:port identifier.

Reading available Requests

Subscribers send their request to your application and Publisher keeps them in a FIFO together with some messages it sends to inform your application about the connection status and the communication between publisher and subscribers. Use **availableRequests** property node to retrieve the number of received request that your application has to process. When a request is processed with **getRequest.vi**, **availableRequests** is decremented by 1.

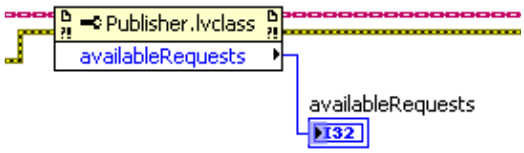


Figure 12 - Using available request property node to check is subscriber's requests are present.

Sending message to a specific subscriber

Your application can communicate with active subscribers with custom messages. To send a message use **sendMessage.vi**, as shown in the following figure:

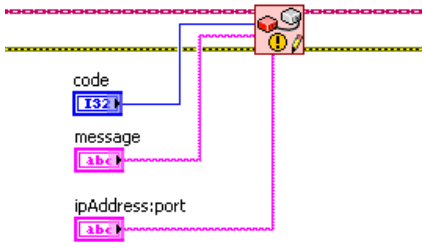


Figure 13 - Sending message to a specific subscriber with sendMessage.vi.

Use **message** string to add additional information to your message. To send a message to every active subscriber **ipAddress:port** must be empty string, instead, if

you want to send a message to a specific subscriber, connect **ipAddress:port** to the specific address and port of its connection. Message string cannot be empty. If try to send an empty string message, error is generated with error code 5002.

Managing User Requests

User request (i.e. message from subscriber) is inserted into a FIFO by Publisher so your application can process all requests in the order they are received from Publisher. The following figure illustrates the right way to manage the requests:

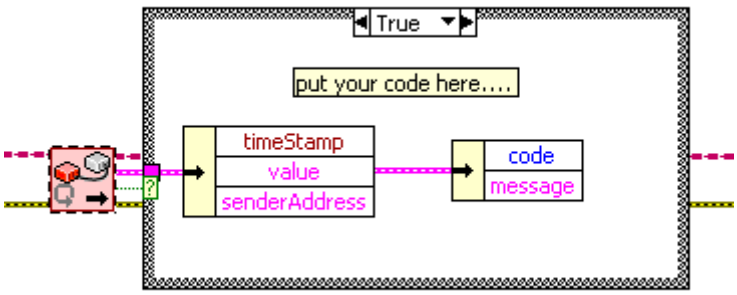


Figure 14 - Managing received request example.

First, you have to check that **found** indicator is TRUE, if **found** is FALSE, no request is available. Every request has its **time** field, **address:port** field that identifies the subscriber that sent the request and an optional **data** field.

Notifying an error

When a LabVIEW error arises and must be notified to one or more subscribers, use notifyError.vi which composes a message with error code and error description from LabVIEW error cluster. The following example shows how to use it. Please note that you can specify a ipAddress:port reference to send error only to a specific listener. When error cluster contains no error, no message is sent.

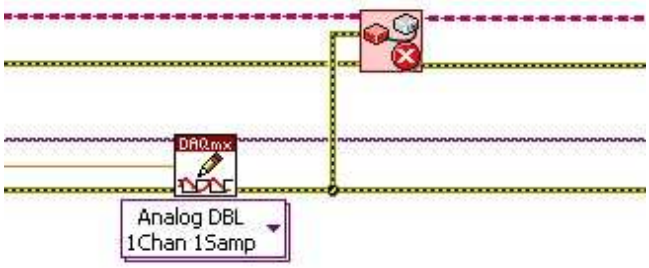


Figure 15 - Notifying LabVIEW error to all subscribers with notifyError.vi.

Advanced Options

Controlling active connections

Publisher Vis handle all incoming connections and close inactive connections so you just focus on your main application and forget all issues related to data transmission. In some cases you want to know the number of active connections and the address of subscribers. Usually you can map all incoming connections using getRequest.vi and filtering events such “connection successful” and “connection closed” and “connection timeout”, which help you to map all active and closed connections.

Reading active connection count

To know the count of active connections at a specific time, use **Read activeSubscribersCount.vi** that returns the number of active connections.

Figure 16 - Use Read activeSubscribersCount.vi to check the count of active connections.

Reading active connection addresses

To know the addresses of active subscribers, use **Read activeSubscribersAddresses.vi that** returns a string array: every string is a subscriber’s address in the form ipAddress:port. the array is ordered by connection time so the first element of the array is related to the active subscribers that connected first. Closed connections don’t appear in the array.

Figure 17 - Use Read activeSubscribersAddresses to get info about active subscribers.

You can use **activeSubscriberCount** property node to get active connection count and **activeSubscriberAddress** property node to get active connection addresses, as shown in the following figure:

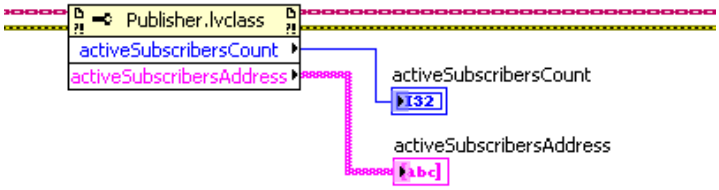


Figure 18 - Use property nodes to get information about active subscribers.

Publishing digital lines on change only

In many cases digital lines don't change very quickly so you can publish them only when they change. in this way you can reduce the consumed band on your communication channel. To change the way publisher works with your digital lines, use the **Digital.publishOnlyOnChange** property node. If you want to publish digital lines also if they aren't changed, set FALSE value to this property node. If you want to publish digital lines only when their value is different from the previous one you published, set this property value to TRUE. By default, a Publisher works with **Digital.publishOnlyOnChange** equal to FALSE:

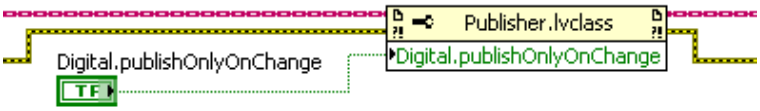


Figure 19 – Changing the way digital data are updated to all subscribers with Digital.publishOnlyOnchange property node.

If your application doesn't need to handle user request you can disable userRequest FIFO to avoid waste of memory. To disable or enable FIFO, use **enableEventLogging** property node, as shown in the next figure.

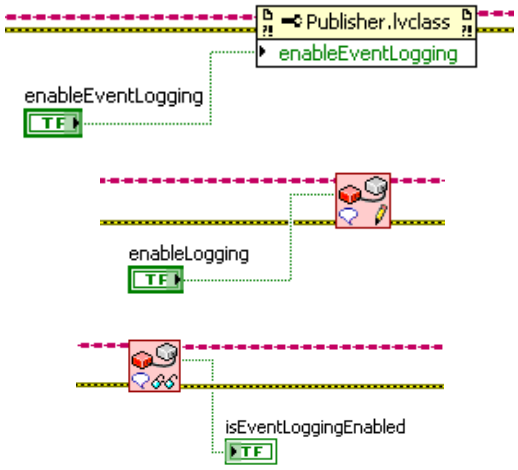


*Figure 20 – Enabling the request FIFO with enableEventLogging property node. Instead of property node you can set the flag "enableLogging" with **enableLogging.vi** and get its value with **isEventLoggingEnabled.vi**, respectively.*

By default, every Publisher starts with FIFO enabled, so remember to consume elements in this FIFO with **getEvent.vi** or disable the FIFO with **enableEventLogging** property node.
Many properties can be set either by specific Vis or by properties nodes. We encourage you to use properties node whenever it is possible.

Managing events

Publisher provides a FIFO used to keep track of communication events. This FIFO is enabled by default and can be enabled / disabled with **eventLogging** property node (see above). Event data cluster contains the following information:

- **timestamp** is the local time when event is occurred.
- **Event** describes the type of event. The available event types are:
    - **authentication successful**, occurs when a new connection is properly established. Is a remote device tries to connect, but its connection parameters are not correct and connection is refused, this event is not generated.
    - **conn closed by client,** occurs when a client terminates connection.
    - **timeout expired,** occurs when a connection is lost abnormally, for example when communication channel fails. In this situation your application cannot be sure that all transmitted packets have been received by client associated to aborted connection.
- **senderAddress** is the address of new subscriber, in the form IpAddress:port.

use **availableEventCount** property node to check is new events have been generated. When a new event is present you can extract it from its event Queue with **getEvent.vi** method as shown in the following figures.

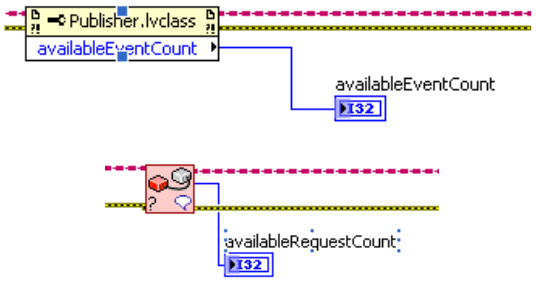


*Figure 21 – Enabling the request FIFO with enableEventLogging property node. Instead of property node you can set the flag "enableLogging" with **enableLogging.vi** and get its value with **isEventLoggingEnabled.vi**, respectively.*

Events are useful when your application keeps track of existing connections and must execute special tasks when a new connection is established or an existing one is closed. Below you see a simple example: a welcome message is sent only to new connected applications. **senderAddress** is the address associated to the connection that generated the event.

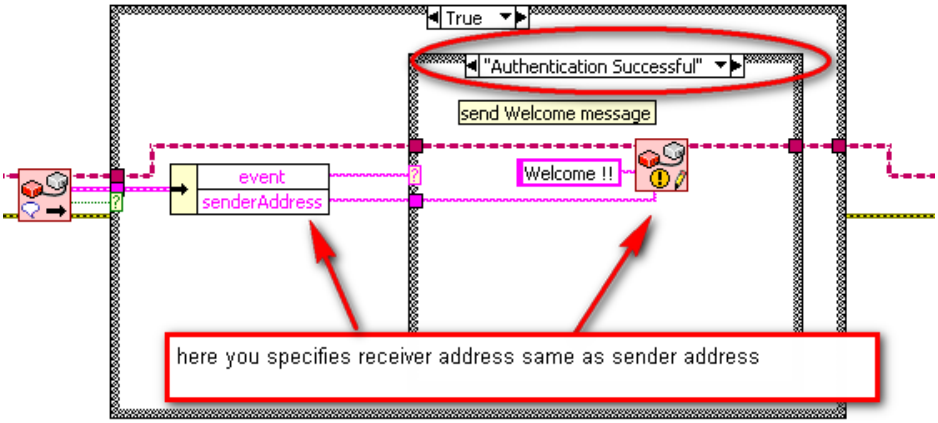


Figure 22 – This example shows how to manage "Authentication Successful" event for instance, Notice that welcome message is addressed to new connected subscriber and other active subscribers are not affected by this message..

In the following figure a different situation is managed: when a connection is terminated, some temporary files associated to it, are deleted from disk. Notice that the case structure uses both "conn closed by client" and "timeout expired" events.

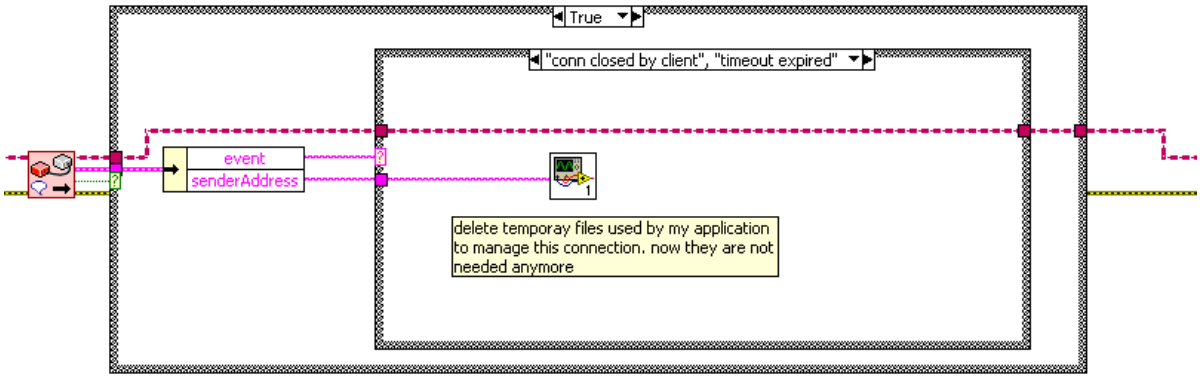

True
"conn closed by client", "timeout expired"
event
senderAddress
delete temporay files used by my application to manage this connection. now they are not needed anymore

Figure 23 – This example shows how to manage the case a connection is no more active. It can happen when client closes the connection or some errors occur during data transfer or timeout, expires.

Changing API-Key at run-time

Publishers are created with an API-Key they use to authenticate every incoming connection. You can modify the API-Key at run-time so that new subscribers have to use the new API-Key to be accepted. Existing connections are not affected by API-Key changes. To update the API-Key use **Write API-Key.vi** as shown in the following figure:

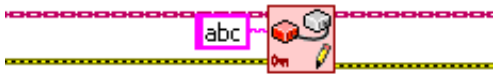


Figure 24 – Changing API-Key to an active Publisher with Write API-Key.vi.

Alternatively you can use API-Key property node as shown below:

Figure 25 – Changing API-Key to an active Publisher using API-Key property node.

Changing max DT

When subscribers try to connect, the request includes their system time. If subscriber's system time differs from publisher's system time more than **maxDt** and **maxDt** is greater than zero, connection is refused. By default, **maxDt** value is zero so subscriber's system time is not checked. To enable time control, use **maxDt** property node to set the absolute value of maximum distance, in seconds, between local and remote system's time.

Figure 26 – Changing maxDt to check subscriber's system time.

Publishing and receiving custom data (supported only by LabVIEW applications)

If both side applications use SCCT for LabVIEW, then some advanced features are available. These features rely on LabVIEW variant data type and their data package are not supported on other programming languages. if more than one custom data type is used, custom data types must be associated to a numeric code or string description so that subscribers can properly identify what data has been sent from publisher. In the following example, publisher side, on the left, encode two different data types and send them with **publishData.vi**. Notice they are associated to code 1 and 2. Subscriber application, on the right, receive the packages with **Read customData.vi**, and uses code field to properly process data type 1 and data type 2.

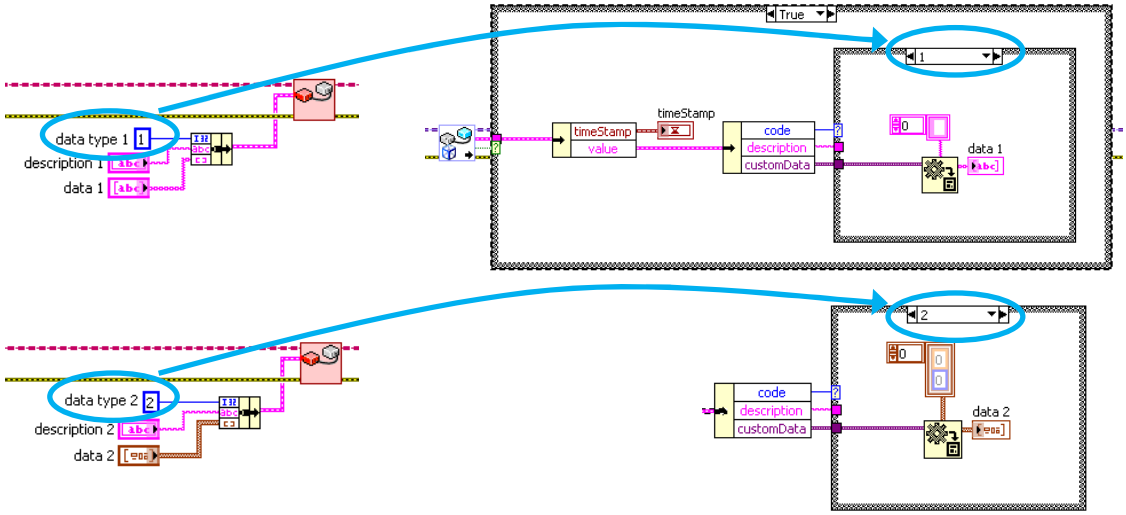


Figure 27 – publisher application and subscriber application use code field to identify the proper data type. Both apps have to know the data type associated to the codes.

Publishing and receiving custom XML data

In addition to messages and requests, both sides of communication channel can exchange XML documents. XML packets have been added to SCCT to allow server and client applications exchanging complex custom data with a single function. Remember that XML documents are not check by SCCT Vis. You must verify XML document's correctness before transmission to avoid unpredictable behaviors by receiving applications. The following figures illustrate the available set of Vis in Publisher and Subscriber's libraries. Notice that on Publisher side.XML transmission is done with the Polymorfic VI **PublishData.vi.**

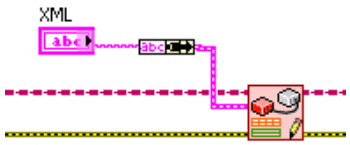


Figure 28 – With Publisher, use publishData.vi method to send XML documents.

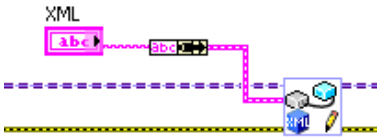


Figure 29 – With Subscriber, use sendCustomXMLData.vi to send XML documents to publisher.

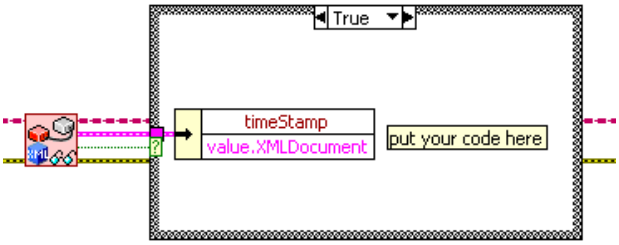


Figure 30 – With Publisher, use Read customXMLData.vi to process XML packets.

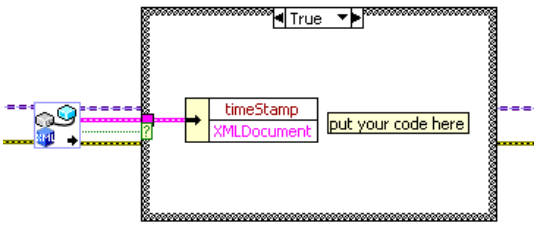


Figure 31 – With Subscriber, use Read customXMLData.vi to process XML packets.

Publisher and Subscriber manage XML document transfer in the same way.

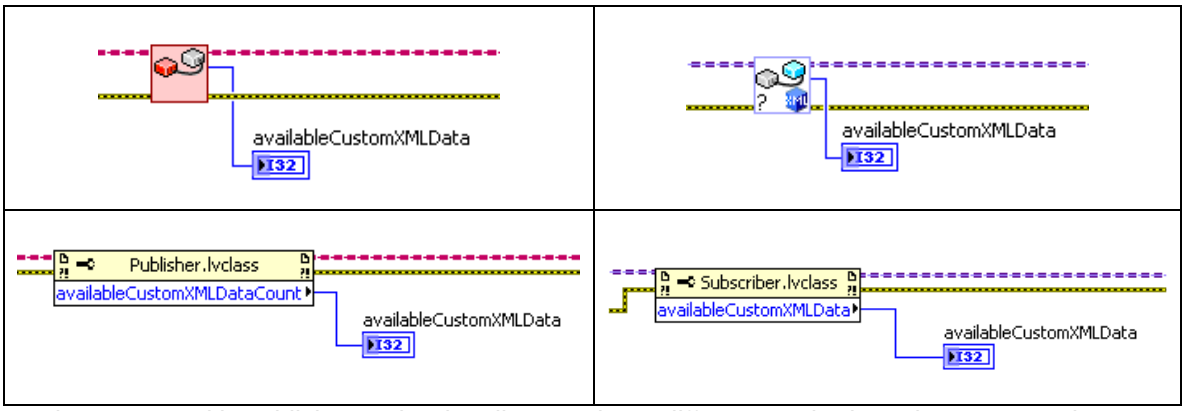


Figure 32 – With Publisher and Subscriber you have different methods and property nodes to get the count of available XML unprocessed documents.

Appendix A - Error table

In the following tables are indicated the error codes generated by VIs

Publisher error codes

code	description	explanation
5000	Publisher is not started	A method or property has been called before creating the Publisher
5001	Invalid port	When you start the publisher you have to provide a valid TCP port.
5002	invalid message text	You cannot send a message with an empty text message
5003	Invalid API-Key	API-Key cannot be an empty string
5004	Invalid configuration	Configuration cluster must contain at least 1 analog channel or 1 digital line
5005	Invalid analog data	Analog data must be a 2D array with exactly the number of rows equal to the number of analog channels as defined in configuration cluster.
5006	Invalid digital Data	Digital data must be a 1D array with size equal to the number of digital lines as defined in configuration cluster
5007	Empty data are not published	Empty arrays are not published
5008	internal library is corrupted	Unexpected error during creation of communication tasks

Subscriber error codes

code	description	explanation
6000	Connection is not established	A method or property has been called before establishing the connection or it has been closed
6001	Invalid port	When you start the publisher you have to provide a valid TCP port.
6002	invalid timeout	Timeout must be a value greater than zero
6003	Invalid API-Key	API-Key cannot be an empty string
6004	Connection cannot be created twice	openConnection.vi cannot be used on an active connection. You have to close a connection before re-open
6005	Connection refused from Publisher: wrong API-Key	Publisher refused the connection because API-Key is not correct
6006	Publisher not found	No answer from specified address: possible reasons: publisher is down or not reachable
6007	Publisher Address cannot be empty	Publisher Address cannot be an empty string

Index

A

active connection
addresses 25
count 25
Analog data
Publish 15; 16
Read 20
API-Key
changing 28
authentication successful 27
available Requests 23

C

Communication
Close 22
Open 20
Custom data
Publish 30
custom XML data 31

D

data transmission
start 19
stop 19
DHCP 20
Digital data
Publish 17

E

Error
Notifying an error 24
error codes
Publisher 33
Subscriber 33
Event
authentication successful 27
connection closed by client 27
managing 27
timeout expired 27

M

Messages
Receiving 21

P

Publisher
Creating 15
methods 14
properties 14
Publisher Class 14
Publishing
Analog Data 15
Custom Data 30; 31
Digital Data 17
Request 21

R

receiving
XML 31
Requests
Sending 21

S

sending
XML 31
start
data transmission 19
stop
data transmission 19
Subscriber
Creating 19
methods 18
properies 18
Subscriber Class 18

T

timeout expired 27

U

User Request 24

X

XML 31
receiving 31
sending 31